# 谷歌 Gemini 2025 GenAI SDK 工具实现方法深度报告：以 Discord 机器人为例

## I. 智能体系统中的函数调用战略作用

函数调用（Function Calling，FC）是大型语言模型（LLM）能力发展中的一个关键里程碑，标志着模型从单纯的文本生成者向能够执行现实世界操作的自治智能体（Agentic System）的转变 1。谷歌 Gemini 2025 SDK 中的工具（Tools）机制，正是实现这一转变的核心协议。

### 1.1. LLM 工具使用的演进：从文本到意图履行

传统的 LLM 交互主要依赖于生成文本回复。然而，随着应用需求的复杂化，模型需要根据用户的自然语言意图，触发外部系统操作，例如检索实时数据、发送电子邮件或控制智能设备。函数调用机制正是弥合自然语言意图和功能代码执行之间的桥梁 1。

与早期 LLM 试图通过提示词（Prompt）引导模型输出特定格式（如 JSON 或 XML）来调用工具的不可靠方法不同，Gemini 提供的原生函数调用协议提供了结构化、可靠的接口。这种结构化方法避免了旧方法中常见的格式错误或模型幻觉（Hallucination），将 LLM 的角色精确定位为一个高级路由和参数生成器。至关重要的是，模型仅负责决定调用哪个函数以及提供所需的参数，**它本身不执行任何代码**。代码的实际执行（例如，调用外部 API 或数据库）始终由应用程序负责，从而确保了操作的安全性和控制权 1。

### 1.2. Gemini 工具协议的先进能力

最新的 Gemini 模型支持多种高级工具执行模式，显著提高了智能体的工作效率和复杂任务的处理能力。

首先是**并行函数调用 (Parallel Function Calling)** 1。模型能够在单个响应中返回多个 FunctionCall 对象。这允许应用程序执行并发的外部请求，例如同时查询当前天气和查找特定股票价格。这种能力对响应延迟（Latency）至关重要。在一个异步框架（如 Discord bot）中，为了最大化并行调用的效率，执行层必须采用完全异步的设计，例如在 Python 中使用 asyncio.gather 或其他并发机制来同时处理多个 I/O 密集型任务。这种架构上的选择是保证机器人在处理复杂、多步骤查询时保持高响应速度的前提。

其次是**组合式函数调用 (Compositional Function Calling)** 1。模型能够在对话的多轮交互中，使用前一个函数调用的结果作为上下文或参数，来建议后续的函数调用。这种顺序执行的能力使得智能体能够处理更复杂、需要链式操作的业务流程。

### 1.3. 工具声明与结构化输出的区别

开发者必须区分函数调用（Function Calling）和结构化输出（Structured Output，通常称为 JSON 模式）。两者都涉及结构化数据，但在用途和工作流上存在根本差异 2。

**函数调用**是一种**两轮**的对话工作流，旨在实现**行动和数据检索**。模型返回的是一个中间步骤的 FunctionCall 意图，应用程序必须执行该意图并将结果反馈给模型进行下一步的合成 1。

**结构化输出**是一种**单轮**过程，旨在进行**数据提取或分类**。例如，通过提供一个响应模式（如 Pydantic 模型或 JSON Schema），模型被约束其最终输出文本必须遵循该蓝图，以便于下游任务解析，但通常不涉及外部代码执行 2。

如果目标只是格式化数据，JSON 模式通常足够。然而，如果目标是使机器人在现实世界中采取行动（例如，实时数据查询或执行操作），则函数调用是强制性的，它要求应用程序必须管理多轮对话状态。

## II. Gemini 函数调用机制技术解析

本节详细阐述 Gemini API 中函数调用的协议细节，包括如何定义工具接口以及执行过程中的四步循环。

### 2.1. 定义工具接口：模式规范与最佳实践

在 Gemini API 中，工具是通过模式（Schema）来定义的。这些模式通常兼容 OpenAPI 规范，详细描述了函数调用的 name（名称）、description（描述）以及 parameters（参数）4。

为了提升开发体验和确保类型安全，谷歌 GenAI SDK 推荐使用语言特定的库来自动生成这些模式。例如，Python SDK 推荐使用 Pydantic，而 JavaScript SDK 推荐使用 Zod 3。通过定义 Python 中的 BaseModel 类，并精确地使用类型注解和 Field(description=...) 来描述参数，SDK 可以自动将这些结构转换为模型可理解的 Tool 对象和 FunctionDeclaration 5。

描述的质量至关重要。模型在决定是否调用函数时，**无法访问**函数内部的实际代码 6。它完全依赖于提供的函数和参数的 description 字段来理解工具的用途和如何填充参数 5。因此，模糊或不准确的描述可能导致模型“幻觉”出不存在的工具或拒绝使用现有工具。

### 2.2. 多轮工具执行循环（四步协议）

函数调用协议是一种结构化的、多轮交互机制，明确定义了应用程序和 Gemini API 之间的通信合约。

以下表格详细说明了函数调用流程的四个核心步骤 1：

Gemini 函数调用协议与对话流程

| **步骤** | **执行者** | **操作描述** | **API 组件/负载** |
| --- | --- | --- | --- |
| 1. 请求 | 应用程序 | 向模型发送用户提示和可用工具（函数声明）列表。 | generate\_content() 调用，附带 tools 配置 1。 |
| 2. 意图/建议 | Gemini 模型 | 分析提示。如果工具相关，返回结构化的 FunctionCall 对象，指定函数名称和参数。 | 响应包含 response.functionCalls 1。 |
| 3. 执行 | 应用程序 | 解析 FunctionCall 对象，路由执行到对应的应用代码，并执行外部 I/O 操作（例如，数据库查询、外部 API 调用）。 | 自定义异步应用逻辑。**这是应用程序的责任** 1。 |
| 4. 合成/响应 | 应用程序与模型 | 将执行结果封装成 FunctionResponsePart 发送回模型。模型利用该数据生成最终、友好的、对话式的用户回复。 | 第二次 generate\_content() 调用，内容历史包含 role='function' 和 FunctionResponsePart 1。 |

需要注意的是，步骤 4 的成功，即模型能够根据工具执行结果生成有意义的最终响应，完全依赖于应用程序准确维护和重新提交完整的对话历史。在步骤 2 和步骤 4 之间，如果应用程序丢失了原始用户提示和模型的中间 FunctionCall 响应，模型将丢失上下文，从而无法完成最终的合成，可能导致重复调用函数或生成失败的回复 1。因此，强大的上下文历史管理是实现可靠函数调用对话的关键。

## III. 架构蓝图：将 Gemini 工具集成到 Discord

将 Gemini 函数调用协议集成到像 Discord 这样的多用户、实时通信平台时，必须解决由平台异步特性和多轮对话状态管理带来的复杂挑战。

### 3.1. Discord 应用架构审查与异步要求

无论是使用 Python 的 discord.py (基于 asyncio) 还是 Node.js 的 discord.js (基于事件循环)，现代 Discord 机器人框架都要求外部操作必须是非阻塞的。

函数调用协议的\*\*步骤 3（执行）\*\*涉及调用外部 API 或数据库，这些操作本质上是 I/O 密集型且通常是同步的。如果在主 Discord 事件循环中同步执行这些阻塞任务，将导致事件循环停滞，机器人将错过心跳信号，最终导致连接中断或卡顿，影响服务器上所有用户的体验 8。

**技术解决方案：** 在 Python 环境中，必须使用 asyncio.to\_thread 或专门的执行器服务（ExecutorService）将阻塞式的函数执行卸载到单独的线程池或进程池中。这确保了主事件循环保持响应性，可以继续处理 Discord 实时事件，同时后台完成 I/O 密集型的工具执行 8。此外，推荐使用 Discord 的应用命令（Slash Commands，/commands），它提供了结构化的输入，相比传统的消息监听器更适合处理工具调用所需的结构化参数 9。

### 3.2. 工具驱动对话的状态管理

在 Discord 这种高并发的多用户环境中，可靠的状态持久化是扩展 Gemini FC 机器人的核心挑战。简单的内存历史存储不足以应对复杂的多轮交互。

**外部上下文存储的必要性：** 为了在多个用户和频道之间保持对话历史，尤其是在 FC 流程的中间状态（步骤 2 到步骤 4 之间），对话内容必须外部化存储，通常以 Discord 用户 ID 或频道 ID 作为键 10。

**混合内存架构：** 生产级应用通常采用混合架构 11：

1. **短期上下文：** 将当前 ChatSession 的内容（消息列表、角色、FunctionCall 状态）存储在高速缓存（如 **Redis**）中，以快速弥合 FC 流程中的两次 API 调用间隔。
2. **长期记忆：** 对于超出 LLM 最大令牌窗口的历史记录，应使用持久性数据库或向量存储。

**缓解令牌限制：** LLM 都有最大输入令牌限制，随着对话历史增长，直接传递所有历史会话会超出限制，并增加成本 11。因此，必须实施策略来管理上下文。一种有效的方法是采用**摘要算法**：定期使用模型对旧消息批次进行总结，然后只向 LLM 发送最新的消息和历史总结。这既能保持关键上下文，又能显著减少令牌消耗和成本 11。

以下表格将 Gemini 函数调用协议步骤与 Discord 机器人的具体架构要求进行映射：

Mapping Gemini FC Steps to Discord Architecture

| **Gemini FC 步骤** | **Discord Bot 行动** | **架构影响** |
| --- | --- | --- |
| 步骤 1 (初始请求) | 事件处理器接收消息，从数据库中检索历史记录，启动 chat.send\_message()。 | 需要状态检索和将完整的工具集配置到 Gemini 客户端.1 |
| 2 (FunctionCall) | Bot 接收到 FunctionCall 对象。 | **关键持久化点：** 必须在继续之前将当前会话状态和待处理的 FunctionCall 参数保存到外部存储 (Redis/DB)。 |
| 3 (执行) | 使用线程池异步执行应用侧函数。 | **非阻塞要求：** 必须将同步 I/O 卸载，以避免 Discord 事件循环崩溃.8 |
| 4 (合成) | 检索保存的状态，将 FunctionResponsePart 提交回 chat.send\_message()，然后回复 Discord 频道。 | 需要状态检索，并保证最终文本响应的成功交付.1 |

## IV. 完整案例分析：基于 Python 的 Gemini 工具 Discord 机器人实现

本案例使用 Python 语言，结合 discord.py 和 google-genai SDK，演示如何实现一个具备工具调用能力的 Discord 机器人。

### 4.1. 环境设置与客户端初始化

首先，开发者需要完成 Discord 应用程序和 Bot 的注册，获取 Bot Token 13，并从 Google AI Studio 获取 Gemini API 密钥 14。所有敏感凭证都应通过环境变量安全加载。

核心依赖库包括 discord.py、google-genai、pydantic 和 python-dotenv。

Python

# 核心依赖库  
import discord  
from google import genai  
from pydantic import BaseModel, Field  
import asyncio  
import os  
  
# 初始化 Discord Client 和 Gemini Client  
DISCORD\_TOKEN = os.getenv("DISCORD\_BOT\_TOKEN")  
GEMINI\_API\_KEY = os.getenv("GOOGLE\_API\_KEY")  
  
gemini\_client = genai.Client(api\_key=GEMINI\_API\_KEY)  
intents = discord.Intents.default()  
intents.message\_content = True # 必须启用消息内容意图  
bot = discord.Client(intents=intents)  
  
# 假设外部存储类 (Redis/DB) 已经实现，用于保存聊天历史  
chat\_sessions = {}

### 4.2. 定义外部工具：库存管理器示例

假设需要构建一个库存管理工具，允许用户查询产品库存或向订单中添加产品。通过 Pydantic 定义函数声明，利用其字段描述向模型传达意图 3。

Python

# 1. 定义 Pydantic Schema (用于自动生成 FunctionDeclaration)  
class ProductQuery(BaseModel):  
 """用于查询产品库存的工具"""  
 product\_id: str = Field(description="要查询库存的产品ID，例如 SKU123。")  
  
class OrderAddition(BaseModel):  
 """用于将产品添加到用户订单的工具"""  
 product\_id: str = Field(description="要添加到订单中的产品ID。")  
 quantity: int = Field(description="要添加的产品数量，必须是正整数。")  
  
# 2. 定义实际的 Python 函数 (应用代码)  
def get\_product\_stock(product\_id: str) -> str:  
 """模拟查询产品库存的外部 API 调用"""  
 # 实际应用中此处应执行 I/O 操作  
 if product\_id == "SKU123":  
 return f"Product SKU123 has 150 units in stock."  
 else:  
 return f"Product {product\_id} not found in inventory."  
  
def add\_product\_to\_order(product\_id: str, quantity: int) -> str:  
 """模拟向订单中添加产品的操作"""  
 # 实际应用中此处应执行 I/O 操作  
 return f"Successfully added {quantity} units of product {product\_id} to the current order."

### 4.3. 核心 Bot 逻辑和聊天管理

当 Discord 机器人接收到一条消息时，它首先需要加载或初始化该会话的 ChatSession，并配置可用工具。工具列表由 Pydantic 模型自动生成。

Python

@bot.event  
async def on\_message(message):  
 if message.author == bot.user:  
 return  
  
 # 仅响应在提到机器人的消息  
 if bot.user in message.mentions:  
 user\_prompt = message.content.replace(f'<@{bot.user.id}>', '').strip()  
 chat\_id = message.channel.id # 使用频道ID作为会话键  
  
 if chat\_id not in chat\_sessions:  
 # 初始化新的聊天会话，并将工具声明传入  
 tools\_list = [ProductQuery, OrderAddition]  
 chat\_sessions[chat\_id] = gemini\_client.chats.create(  
 model="gemini-2.5-flash",   
 tools=tools\_list  
 )  
   
 chat = chat\_sessions[chat\_id]  
   
 # 步骤 1: 发送用户提示和工具列表  
 response = await chat.send\_message\_async(user\_prompt)  
 await handle\_gemini\_response(message, chat, response)

### 4.4. 执行阶段：处理函数调用和响应（Agentic 核心）

这是函数调用循环的核心部分，处理模型的中间响应，执行工具，并将结果反馈给模型进行最终合成。

Python

async def handle\_gemini\_response(discord\_message, chat\_session, response):  
 # 步骤 2: 检查模型响应是否包含 FunctionCall 意图   
 if response.function\_calls:  
 print(f"Model requested function calls: {response.function\_calls}")  
   
 function\_responses =  
 # 使用 asyncio.gather 来支持并行执行多个函数调用  
 tasks =  
   
 for call in response.function\_calls:  
 # 步骤 3.1: 映射 FunctionCall 名称到实际 Python 函数  
 func\_to\_call = globals().get(call.name)  
 if func\_to\_call:  
 # 步骤 3.2: 使用 asyncio.to\_thread 卸载同步 I/O 任务  
 # 避免阻塞 Discord 事件循环  
 tasks.append(asyncio.to\_thread(func\_to\_call, \*\*dict(call.args)))  
 else:  
 # 处理未定义的函数  
 print(f"Error: Function {call.name} not defined.")  
 tasks.append(asyncio.to\_thread(lambda: "Error: Function not implemented."))  
  
 # 异步等待所有工具执行完成  
 results = await asyncio.gather(\*tasks)  
  
 # 步骤 3.3: 格式化函数执行结果  
 for call, result in zip(response.function\_calls, results):  
 function\_responses.append(  
 genai.types.Part.from\_function\_response(  
 name=call.name,  
 response={"result": result} # 将结果封装  
 )  
 )  
  
 # 步骤 4: 将函数结果作为 FunctionResponsePart 发送回模型  
 # 注意：SDK 自动管理历史记录的重新提交  
 print("Sending function results back to Gemini for synthesis...")  
 final\_response = await chat\_session.send\_message\_async(  
 function\_responses,  
 # 将 FunctionResponsePart 数组作为消息内容发送  
 )  
   
 # 最终交付  
 await discord\_message.reply(final\_response.text)  
  
 # 如果模型直接返回了文本  
 elif response.text:  
 await discord\_message.reply(response.text)  
  
 # 极少数情况下，模型无响应或返回空内容  
 else:  
 await discord\_message.reply("Sorry, the AI was unable to generate a useful response.")

## V. 高级实践与运营化

将一个具备函数调用能力的 Discord 机器人投入生产环境，需要考虑鲁棒性、可扩展性和安全性等高级问题。

### 5.1. 鲁棒的错误处理和拒绝协议

在实际部署中，外部工具执行失败是常见情况（例如，API 返回 404 错误、数据库连接超时）。如果执行控制器默默地处理了这些错误，模型在步骤 4 中将无法获得正确的信息，可能导致最终响应混乱。

**向 LLM 报告错误：** 健壮的系统要求，如果函数执行失败，应用程序必须明确地将错误详细信息封装在 FunctionResponsePart 结构中，并发送回 Gemini 模型 1。例如，可以将 result 字段设置为包含错误堆栈或 API 错误消息的字符串。这使得模型能够理解工具执行失败的原因，并以对话的方式向用户解释，而不是简单地抛出错误或沉默。

**工具拒绝处理：** 系统需要确保在模型判定可用工具与用户提示无关或不足以回答问题时，能够优雅地生成一个直接的文本回复，而不是陷入不必要的执行循环。在上述案例中，通过检查 response.function\_calls 和 response.text 的存在性，实现了这种分支逻辑。

### 5.2. 规模化与性能优化

对于大型 Discord 服务器，性能和成本管理至关重要。

**延迟管理：** 函数调用协议本质上是两次 LLM 调用（步骤 1 和 4）加上一次外部 I/O 操作（步骤 3）。为了最小化总延迟，**并行函数调用**是关键的优化手段 1。同时，通过优化外部工具的 I/O 性能，并确保 Discord 机器人的异步架构不会被阻塞，可以显著提高用户体验。

**成本控制与令牌管理：** 在处理长对话时，必须采用自动化的上下文修剪和摘要技术 11。通过将历史记录存储在外部存储中并仅向 LLM 传递相关的最新消息和摘要，可以确保对话保持在令牌窗口内，从而有效控制 API 成本 11。

### 5.3. 安全与信任边界

集成函数调用极大地改变了系统的安全模型。由于应用程序是根据 LLM 生成的参数来执行**本地代码**的，安全边界从用户输入转移到了 LLM 的输出。

**LLM 参数的输入验证：** 尽管模型返回的 FunctionCall 对象结构上符合预定义的 Pydantic/OpenAPI 模式，但其内容（参数值）来自一个非确定性的、基于文本的生成过程。如果工具与敏感系统（如文件系统、数据库）交互，恶意用户可能通过提示注入（Prompt Injection）尝试诱导 LLM 生成具有破坏性的参数（例如，在文件路径参数中尝试注入 ../../etc/passwd）。

因此，即使参数符合类型定义，应用程序层也必须对从 FunctionCall 中提取的所有参数执行严格的运行时验证和清理。应用程序应将 LLM 的输出视为一个高级建议，而不是绝对的、无条件的执行命令 15。这是保护后端系统免受 LLM 驱动的攻击的必要安全措施。

## VI. 总结与建议

Gemini 2025 GenAI SDK 提供的函数调用能力，为在 Discord 等平台上构建高度智能、能够执行真实世界任务的机器人提供了强大的基础。其核心机制在于一个结构化的四步协议，将模型的意图生成与应用程序的代码执行清晰地分离。

**核心结论与建议：**

1. **强制采用异步架构：** 在 Discord 环境中实现函数调用，必须使用 asyncio.to\_thread 等机制将 I/O 密集型工具执行从主事件循环中剥离，以保证机器人的高响应性和稳定性。
2. **实施外部状态管理：** 必须依赖 Redis 或数据库等外部存储来持久化多轮对话状态，尤其是在 Function Calling 的中间步骤，确保模型能够维持上下文并完成最终的响应合成。
3. **高质量的工具描述至关重要：** 模型完全依赖 FunctionDeclaration 中的描述来理解和使用工具。开发者应投入精力编写清晰、详细的函数和参数描述，并利用 Pydantic 等库确保模式的类型安全。
4. **将 LLM 输出视为输入：** 在安全敏感的应用中，开发者必须将从 FunctionCall 中提取的参数视为来自外部的、不可信的输入，并在代码执行前进行严格的运行时验证和清理，以防止潜在的命令注入或滥用。

#### Works cited

1. Function calling with the Gemini API | Google AI for Developers, accessed on November 6, 2025, <https://ai.google.dev/gemini-api/docs/function-calling>
2. Generate structured output (like JSON and enums) using the Gemini API | Firebase AI Logic, accessed on November 6, 2025, <https://firebase.google.com/docs/ai-logic/generate-structured-output>
3. Structured Outputs | Gemini API - Google AI for Developers, accessed on November 6, 2025, <https://ai.google.dev/gemini-api/docs/structured-output>
4. Introduction to function calling | Generative AI on Vertex AI - Google Cloud Documentation, accessed on November 6, 2025, <https://docs.cloud.google.com/vertex-ai/generative-ai/docs/multimodal/function-calling>
5. How to Interact with APIs Using Function Calling in Gemini, accessed on November 6, 2025, <https://codelabs.developers.google.com/codelabs/gemini-function-calling>
6. Day 3 - Function calling with the Gemini API - Kaggle, accessed on November 6, 2025, <https://www.kaggle.com/code/markishere/day-3-function-calling-with-the-gemini-api>
7. Function calling using the Gemini API | Firebase AI Logic - Google, accessed on November 6, 2025, <https://firebase.google.com/docs/ai-logic/function-calling>
8. API Reference - Discord.py, accessed on November 6, 2025, <https://discordpy.readthedocs.io/en/latest/api.html>
9. Application Commands | Documentation | Discord Developer Portal, accessed on November 6, 2025, <https://discord.com/developers/docs/interactions/application-commands>
10. Integrating AI Into a Discord Bot: Ultimate Guide, accessed on November 6, 2025, <https://cybrancee.com/blog/integrating-ai-into-a-discord-bot-ultimate-guide/>
11. Managing Chat History at scale in Generative AI Chatbots | AWS Builder Center, accessed on November 6, 2025, <https://builder.aws.com/content/2j9daS4A39fteekgv9t1Hty11Qy/managing-chat-history-at-scale-in-generative-ai-chatbots>
12. Check out my AI-Powered Discord Bot using Python and OpenAI API - Roast, Summarize, Visualize, and more! - Reddit, accessed on November 6, 2025, <https://www.reddit.com/r/Python/comments/1319ftx/check_out_my_aipowered_discord_bot_using_python/>
13. Gemini has entered the chat: building an LLM-powered Discord bot | by Maciej Strzelczyk | Google Cloud - Community | Medium, accessed on November 6, 2025, <https://medium.com/google-cloud/gemini-has-entered-the-chat-building-an-llm-powered-discord-bot-872f876d92f8>
14. hihumanzone/Gemini-Discord-Bot: A Discord bot ... - GitHub, accessed on November 6, 2025, <https://github.com/hihumanzone/Gemini-Discord-Bot>
15. Function calling with chat completion | Microsoft Learn, accessed on November 6, 2025, <https://learn.microsoft.com/en-us/semantic-kernel/concepts/ai-services/chat-completion/function-calling/>